Lecture 3: Simple Sorting and Searching Algorithms

Data Structure and Algorithm Analysis

Searching and Sorting

<u>Topics</u>

- Searching
  - Linear/Sequential Search
  - Binary Search
- Sorting
  - Bubble Sort
  - Insertion Sort
  - Selection sort

Common Problems

- There are some very common problems that we use computers to solve:
  - **Searching**: Looking for specific data item/record from list of data items or set of records.
  - **Sorting :** placing records/items in order

- There are numerous algorithms to perform searches and sorts.
- We will briefly explore a few common ones in this lecture.

Searching

- There exists many searching algorithms you can choose from
- A question you should always ask when selecting a search algorithm is
  - "How fast does the search have to be?"
- Facts
  - In general, the faster the algorithm is, the more complex it is.
  - You don't always need to use or should use the fastest algorithm.
  - The list to be searched can either be ordered or unordered list
- Let's explore the following search algorithms, keeping speed in mind.
  - Sequential (linear) search
  - Binary search

Linear/Sequential Search on an Unordered List

- Basic algorithm:

  Get the search criterion (**key**)

  Get the first record from the file

  While ((record != key) and (still more records))

    Get the next record

  End_while

- When do we know that there wasn't a record in the List that matched the key?

Linear Search (Sequential Search)

- Example Implementation:

```
int linear_search(int list[], int n, int key){
 for (int i=0;i<n; i++){
        if(list[i]==key)
        return i;
 }
 return -1;
}
```

Time complexity O(n)

--Unsuccessful search --- n times

--Successful search (worst) --- n times

--Successful search (Best) --- 1 time

--Successful search (average) --- n/2 times

Sequential Search of Ordered vs. Unordered List

- If sequential search is used on list of integers say [14,80,39,100,-8], how would the search for 100 on the ordered list compare with the search on the unordered list?
  - Unordered list <14,80,39,100,-8>
    - if 100 was in the list?
    - if -50 was not in the list?
  - Ordered list <-8,14,39,80,100>
    - if 100 was in the list?
    - if -50 was not in the list?`

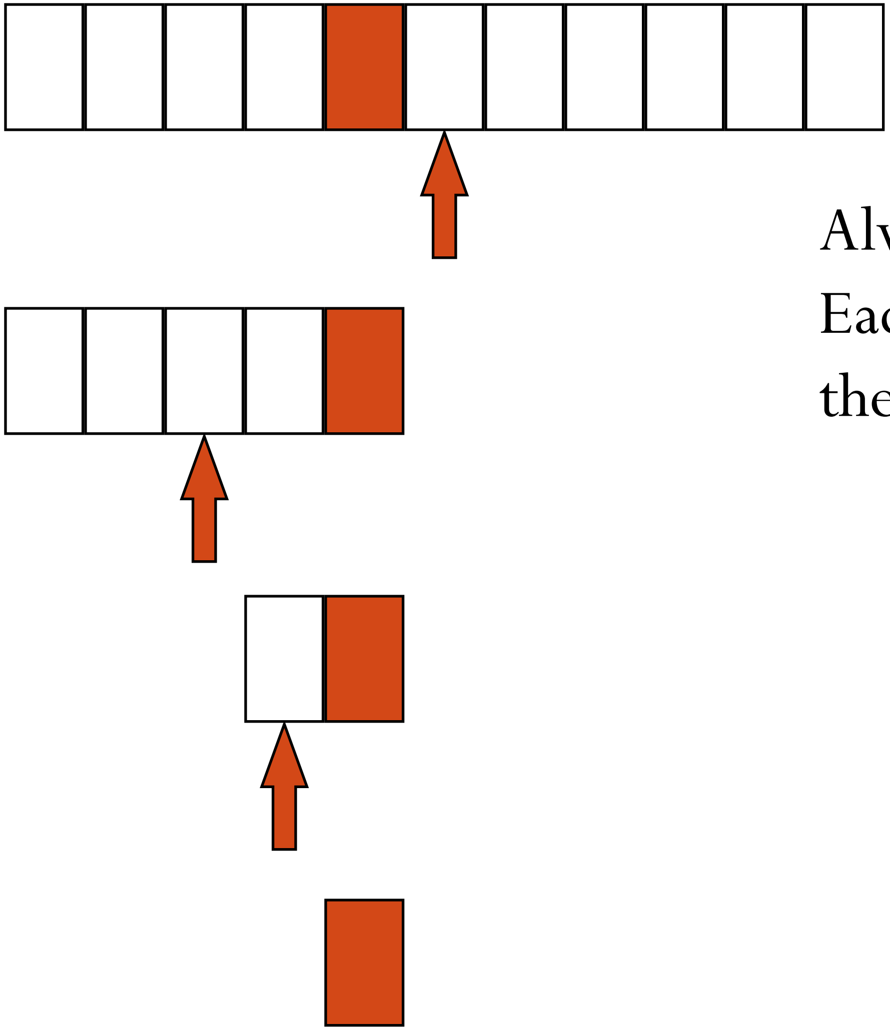
Ordered vs. Unordered (con't)

- **Observation**: the search is faster on an ordered list only when the item being searched for is not in the list.
- Also, keep in mind that the list has to first be placed in order for the ordered search.
- **Conclusion**: the **efficiency** of these algorithms is roughly the same.
- So, if we need a faster search, on sorted list we need a completely different algorithm.

Binary Search

- Sequential search is not efficient for large lists because, on average, the sequential search searches half the list.
- If we have an ordered list and we know how many things are in the list, we can use a different strategy.
- The **binary search** gets its name because the algorithm continually divides the list into two parts.
  - Uses the divide-and-conquer technique to search the list

How a Binary Search Works



Always look at the center value.
Each time you get to discard half of the remaining list.

Is this fast ?

Example Implementation

```
int binary_search(int list[],int n, int key)
{
        int left=0;  int right=n-1;
        int mid;
        while(left<=right){
                        mid=(left+right)/2;
                if(key==list[mid])
                        return mid;
                else if(key > list[mid])
                        left=mid+1;
                else
                        right=mid-1;
        }
        return -1;
}
```

How Fast is a Binary Search?

- Worst case: 11 items in the list took 4 tries
- How about the worst case for a list with 32 items ?
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item

How Fast is a Binary Search? (con't)

List has 250 items

1st try - 125 items

2nd try - 63 items

3rd try - 32 items

4th try - 16 items

5th try - 8 items

6th try - 4 items

7th try - 2 items

8th try - 1 item

List has 512 items

1st try - 256 items

2nd try - 128 items

3rd try - 64 items

4th try - 32 items

5th try - 16 items

6th try - 8 items

7th try - 4 items

8th try - 2 items

9th try - 1 item

Efficiency

- Binary search is one of the fastest Algorithms
- The computational time for this algorithm is proportional to $\log_2 n$
- Lg n means the log to the base 2 of some value of n.
  - $8 = 2^3$ $\lg 8 = 3$ $16 = 2^4$ $\lg 16 = 4$
- Therefore, the time complexity is O(logn)

Sorting

- The binary search is a very fast search algorithm.
  - But, the list has to be sorted before we can search it with binary search.
- To be really efficient, we also need a fast sort algorithm.

- There are many known sorting algorithms.

Bubble Sort	Heap Sort
Selection Sort	Merge Sort
Insertion Sort	Quick Sort

Common Sort Algorithms

- Bubble sort is the slowest, running in **n^2 time**. Quick sort is the fastest, running in **n lg n time**.
- As with searching, the faster the sorting algorithm, the more complex it tends to be.
- We will examine three sorting algorithms:
  - Bubble sort
  - Insertion sort
  - Selection sort

Bubble Sort

- Suppose we have an array of data which is unsorted:
  - Starting at the front, traverse the array, find the largest item, and move (or *bubble*) it to the top

  - With each subsequent iteration, find the next largest item and *bubble* it up towards the top of the array
- Bubble sort is a simple algorithm with:
  - a memorable name, and
  - a simple idea
- It is an O(n2) algorithm and usually called **"the generic bad algorithm"**

Implementation

- Starting with the first item, assume that it is the largest

- Compare it with the second item:
  - If the first is larger, swap the two,
  - Otherwise, assume that the second item is the largest

- After one pass, the largest item must be the last in the list

  Start at the front again:
  - the second pass will bring the second largest element into the second last position
- Repeat $n - 1$ times, after which, all entries will be in place

Bubble Sort Code

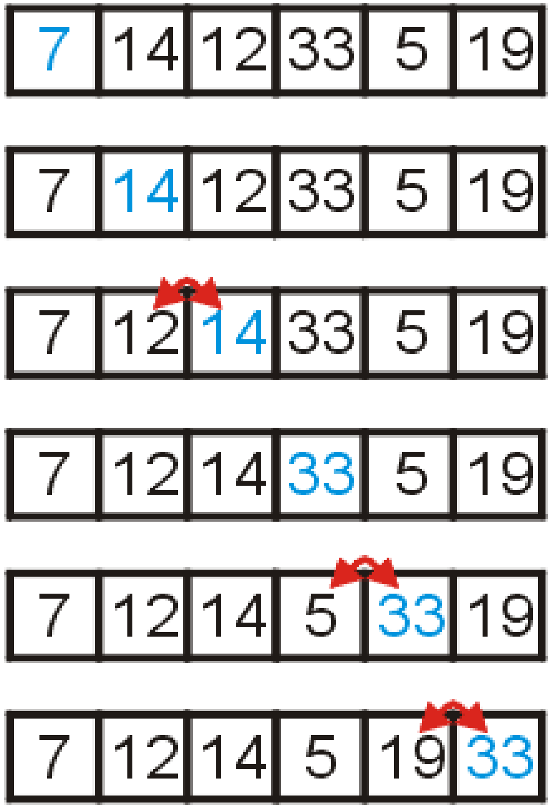
```
void bubbleSort (int a[ ] , int size)
{
   int i, j, temp;
   for ( i = 0; i < size; i++ )    /* controls passes through the list */
   {
           for ( j = 0; j < size - 1; j++ )   /* performs adjacent comparisons */
           {
                   if ( a[ j ] > a[ j+1 ] )   /* determines if a swap should occur */
                   {
                           temp = a[ j ];      /* swap is performed */
                           a[ j ] = a[ j + 1 ];
                           a[ j+1 ] = temp;
                   }
           }
   }
}
```

Example

Consider the unsorted array to the right

We start with the element in the first location, and move forward:

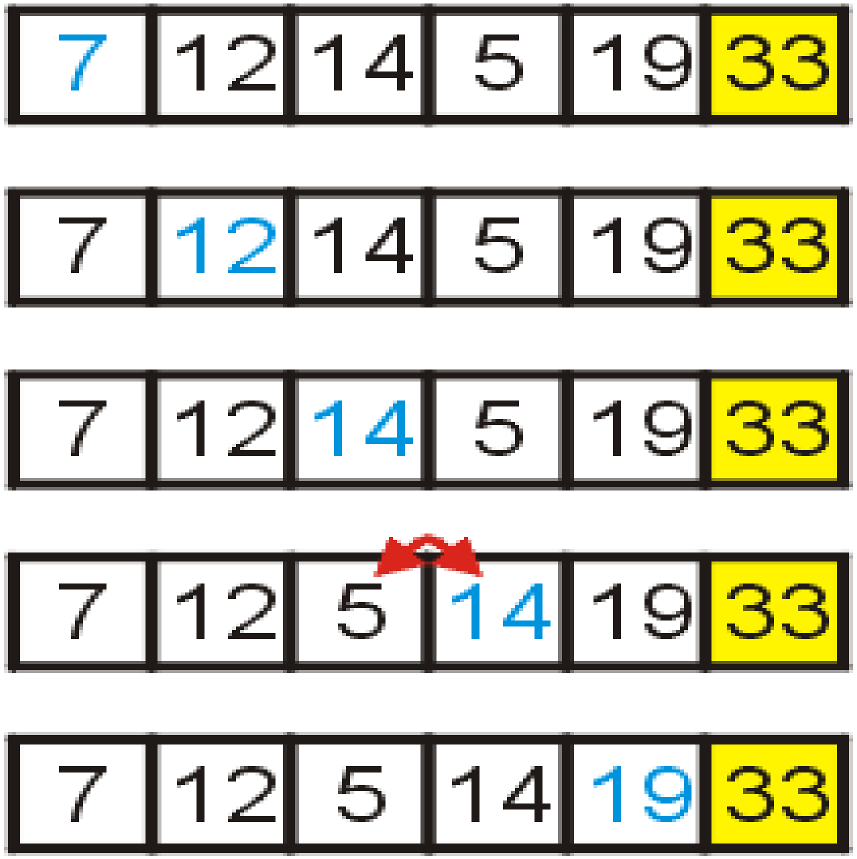
- if the current and next items are in order, continue with the next item, otherwise
- swap the two entries



Example

After one loop, the largest element is in the last location

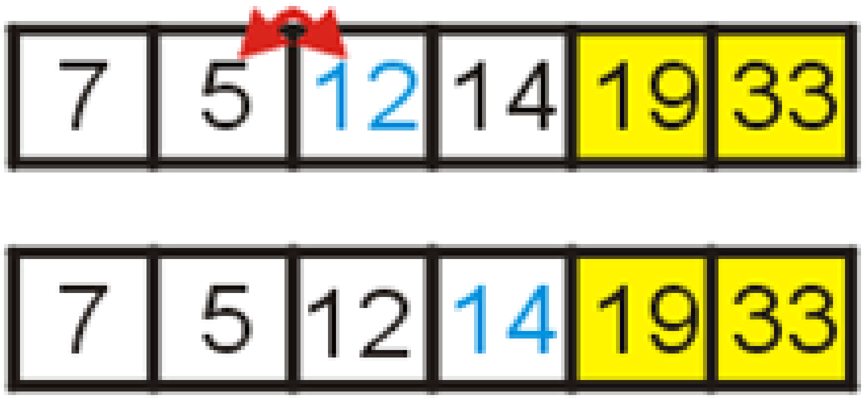
- Repeat the procedure



Example

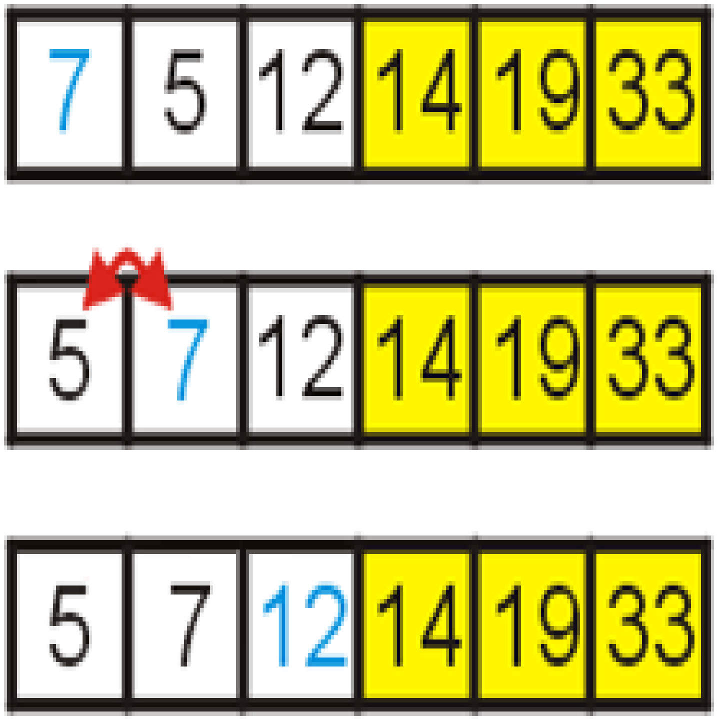
Now the two largest elements are at the end

- Repeat again



Example

With this loop, 5 and 7 are swapped



Example

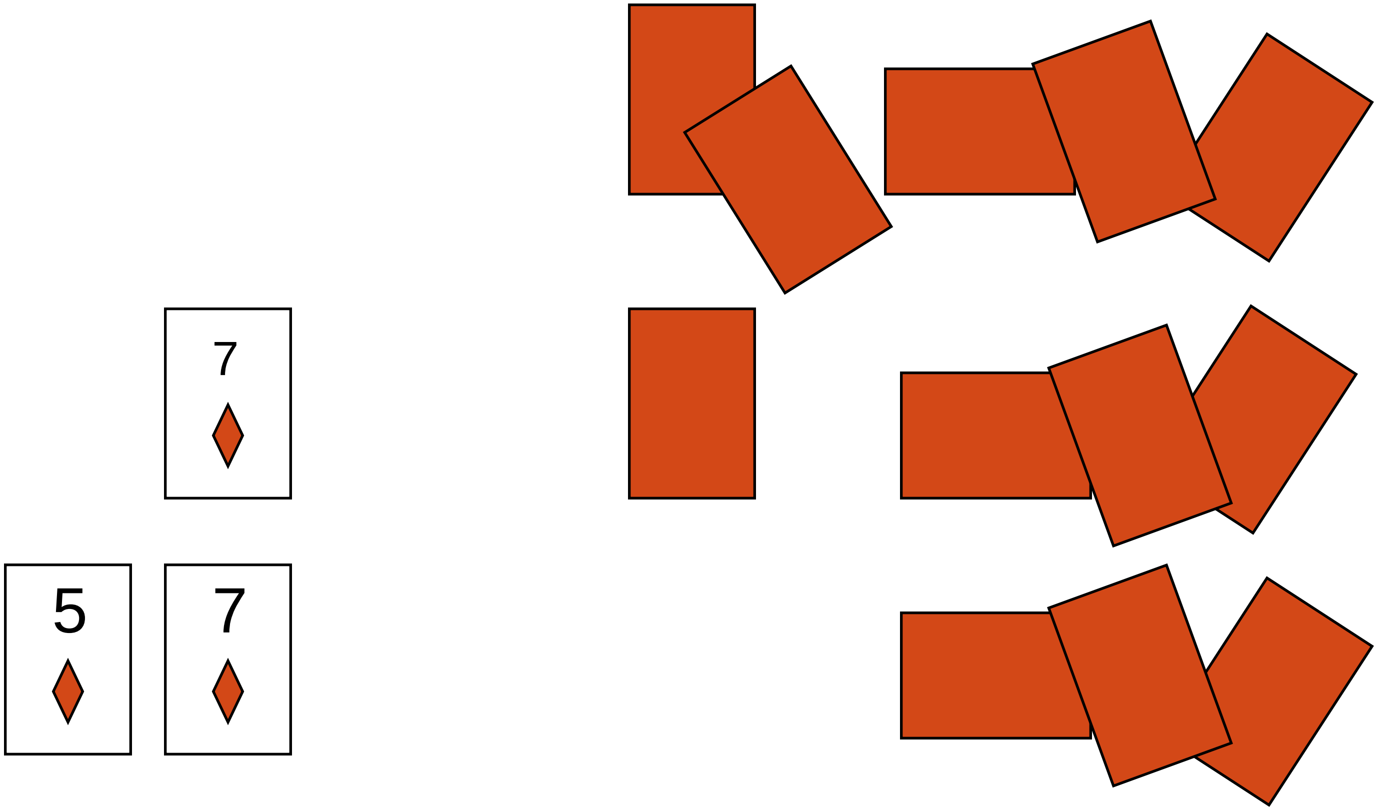
Finally, we swap the last two entries to order them

- At this point, we have a sorted array

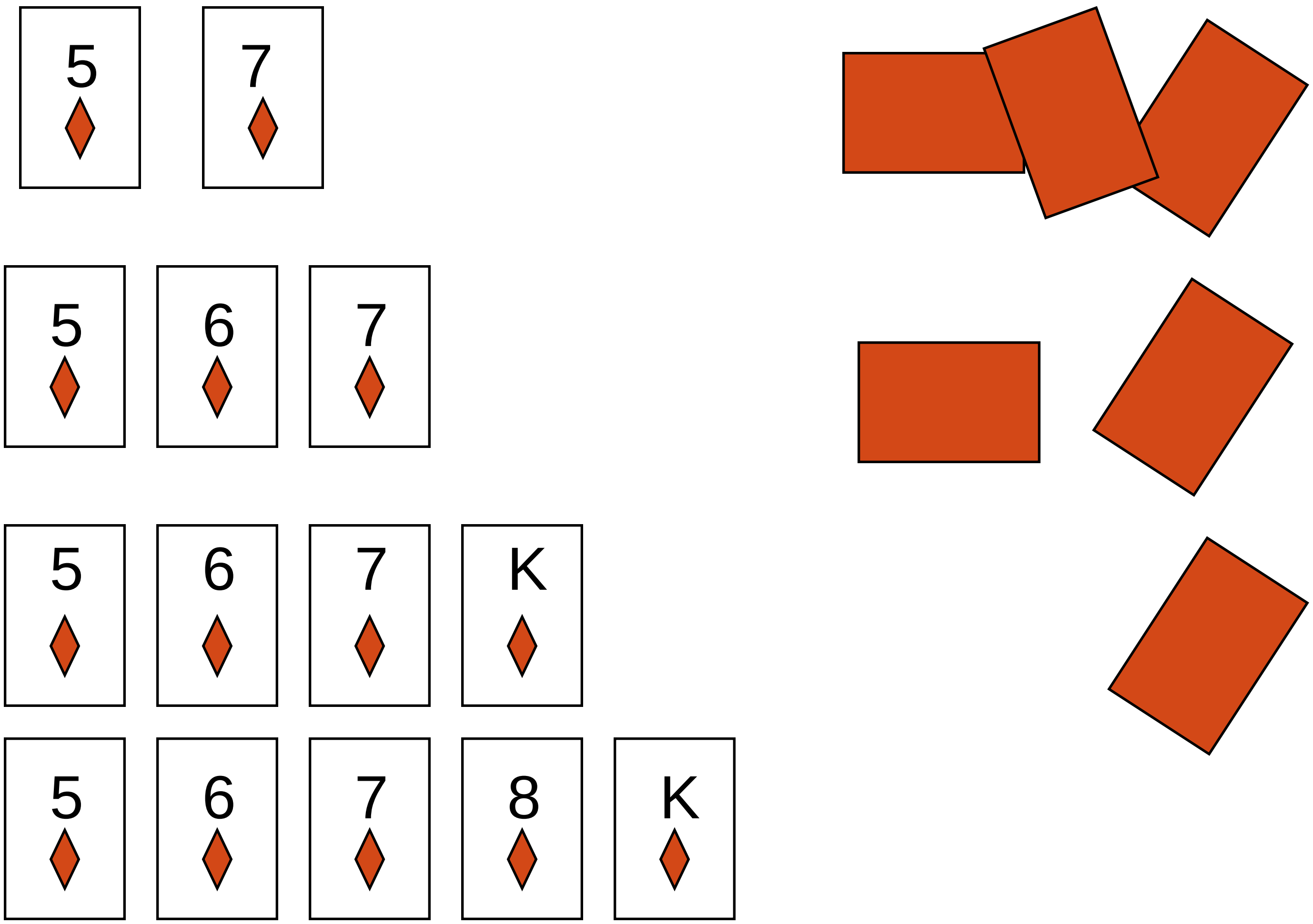
Insertion Sort

- Consider the following observations:
  - A list with one element is sorted
  - In general, if we have a sorted list of k items, we can insert a new item to create a sorted list of size $k + 1$
- Insertion sort works the same way as arranging your hand when playing cards.
  - Out of the pile of unsorted cards that were dealt to you, you pick up a card and place it in your hand in the correct position relative to the cards you're already holding.

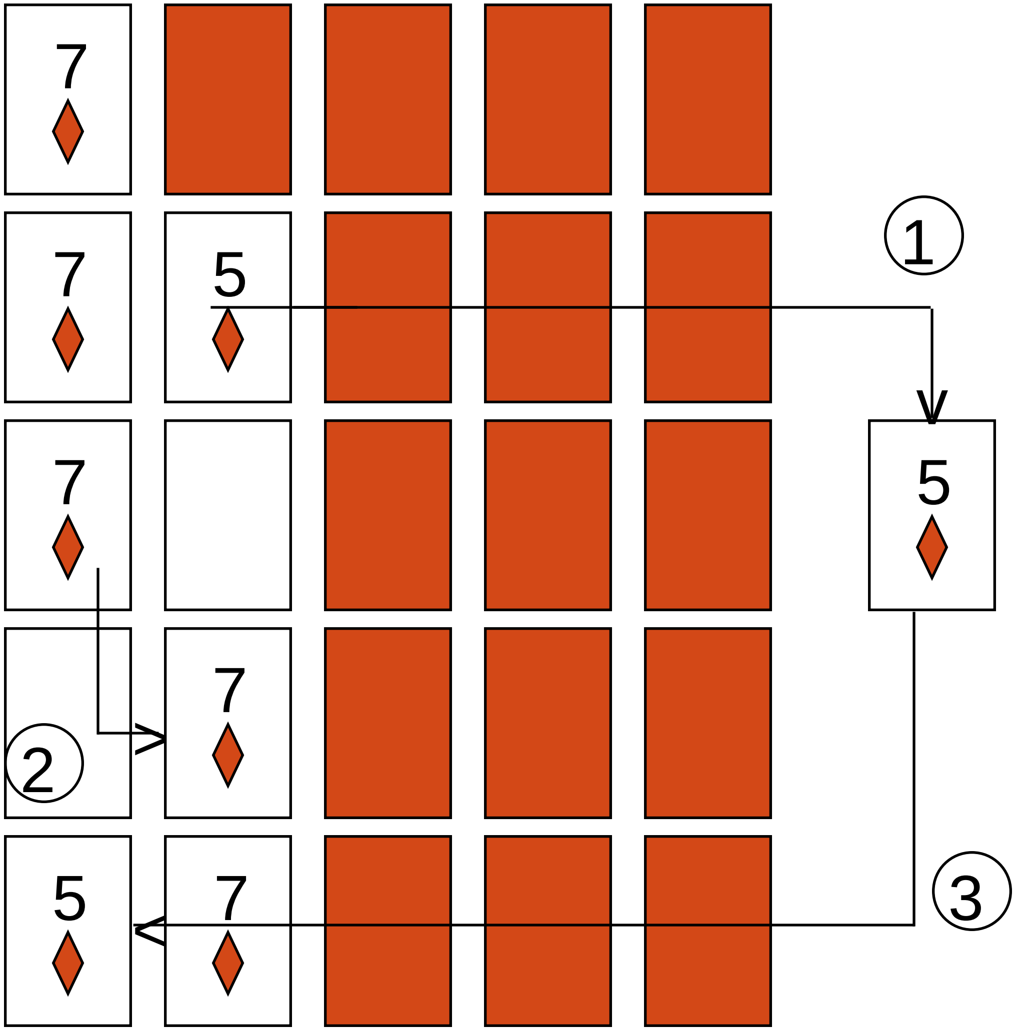
Arranging Your Hand



Arranging Your Hand



Insertion Sort



Unsorted - shaded

Look at 2nd item - 5.

Compare 5 to 7.

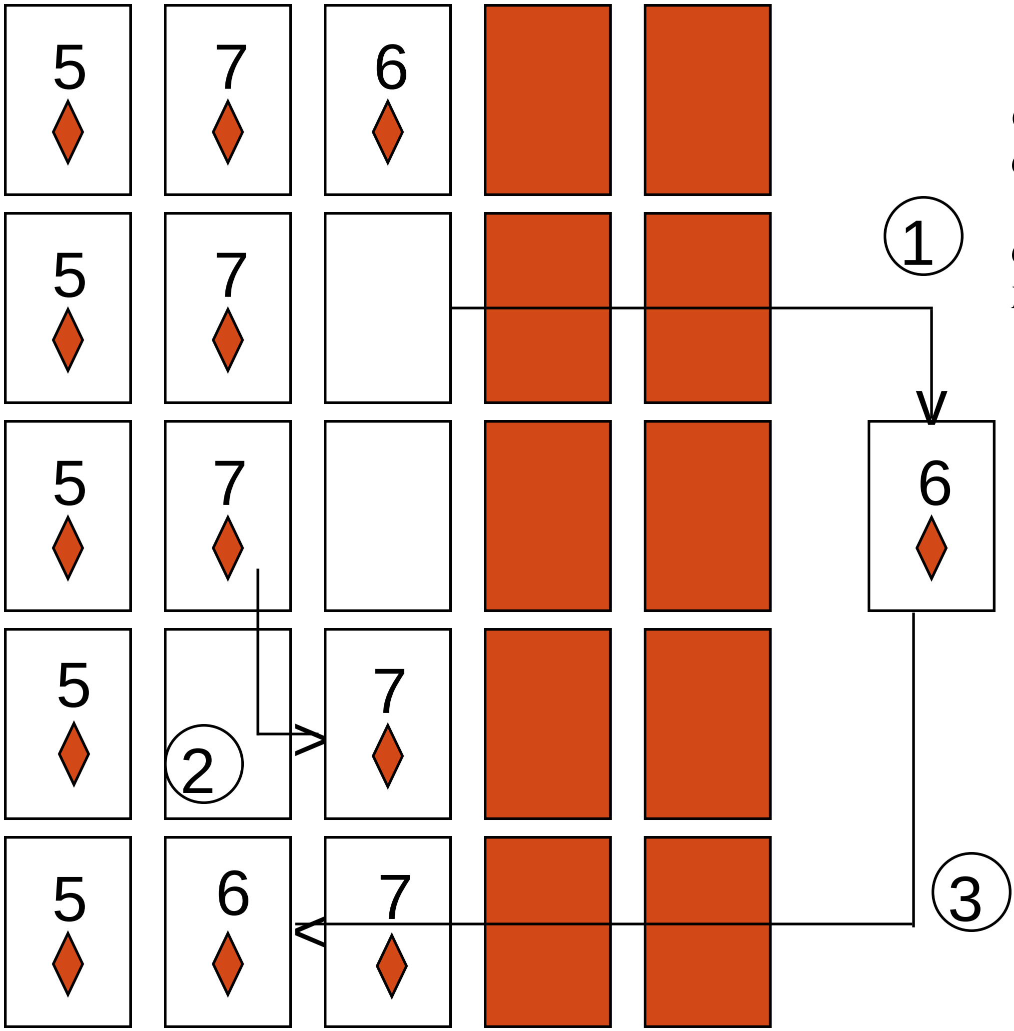
5 is smaller, so move 5 to temp, leaving an empty slot in position 2.

Move 7 into the empty slot, leaving **position 1 open**

Move 5 into the open position

.

Insertion Sort (con't)



Look at next item - 6.

Compare to 1st - 5.
6 is larger, so leave 5. Compare to next - 7.

6 is smaller, so move 6 to temp, leaving an empty slot.
Move 7 into the empty slot, leaving position 2 open.

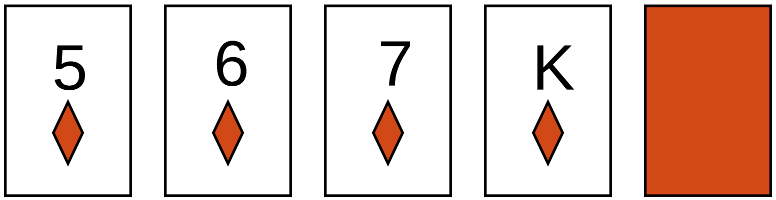
Move 6 to the open 2nd position.

Insertion Sort (con’t)

Look at next item - King.

Compare to 1st - 5.

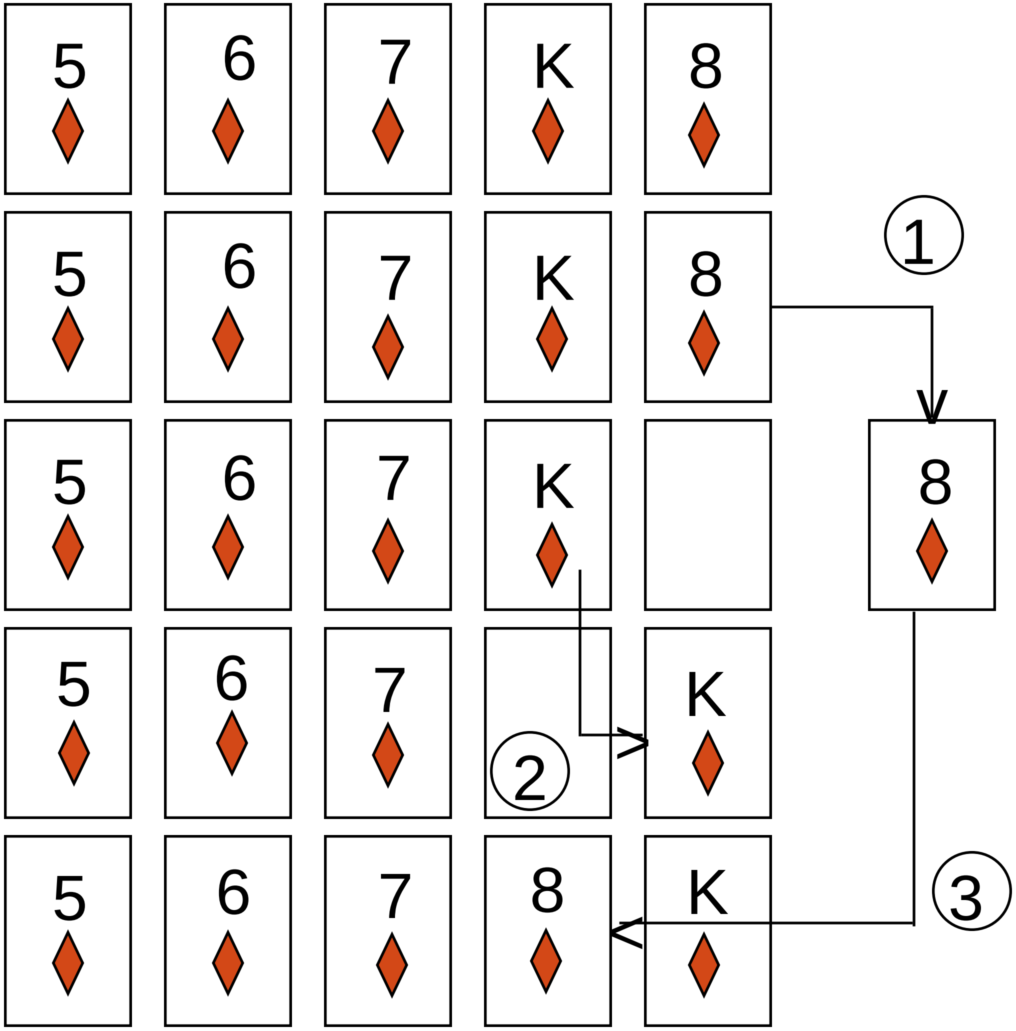
King is larger, so leave 5 where it is.



Compare to next - 6. King is larger, so leave 6 where it is.

Compare to next - 7. King is larger, so leave 7 where it is.

Insertion Sort (con't)



Implementation- Insertion sort

- Basic Idea is:
  - Find the location for an element and move all others up, and insert the element.
- Steps:

1. The left most value can be said to be sorted relative to itself. Thus, we don’t need to do anything.
2. Check to see if the second value is smaller than the first one.
   - ✓ If it is swap these two values.
   - ✓ The first two values are now relatively sorted.
3. Next, we need to insert the third value in to the relatively sorted portion
   - ✓ So that after insertion, the portion will still be relatively sorted.
4. Now the first three are relatively sorted.
5. Do the same for the remaining items in the list.

Implementation- Insertion sort

```
void insertion_sort(int list[ ])
{
int temp;
for(int i = 1; i < n; i++){
    temp = list[i];
for(int j = i; j > 0 && temp < list[j - 1]; j--)
{ //work backwards through the array finding where temp should go
        list[j] = list[j - 1];
        list[j - 1] = temp;
  }//end of inner loop
 }//end of outer loop
}//end of insertion_sort
```

Analysis – Insertion sort

- How many comparisons?

  $1 + 2 + 3 + \ldots + (n-1) = O(n^2)$

- How many swaps?

  $1 + 2 + 3 + \ldots + (n-1) = O(n^2)$

- How much space?

  In-place algorithm

Selection Sort

- **Basic Idea:**
  - Loop through the array from I = 0 to n - 1.
  - Select the smallest element in the array from i to n
  - Swap this value with value at position i.

Implementation- Selection Sort

```
void selection_sort(int list[])
{
int i, j, smallest;
for(i = 0; i < n; i++){
smallest = i;
        for(j = i + 1; j < n; j++){
                if(list[j] < list[smallest])
                        smallest = j;
        }//end of inner loop
        temp = list[smallest];
        list[smallest] = list[i];
        list[i] = temp;
} //end of outer loop
}//end of selection_sort
```

Analysis- Selection Sort

- How many comparisons?

  $(n-1) + (n-2) + \ldots + 1 = O(n^2)$

- How many swaps?

  $n = O(n)$

- How much space?

  In-place algorithm

End of Lecture 3

Next Lecture: Linked lists